Simulating MPI applications: the SMPI approach

Augustin Degomme, Arnaud Legrand, Georges Markomanolis, Martin Quinson, Mark Lee Stillwell, Frédéric Suter

Simulating MPI applications: the SMPI approach

Augustin Degomme, Arnaud Legrand, George S. Markomanolis,
Martin Quinson, Mark Stillwell, and Frédéric Suter

Abstract—This article summarizes our recent work and developments on SMPI, a flexible simulator of MPI applications. In this tool, we took a particular care to ensure our simulator could be used to produce fast and accurate predictions in a wide variety of situations. Although we did build SMPI on SimGrid whose speed and accuracy had already been assessed in other contexts, moving such techniques to a HPC workload required significant additional effort. Obviously, an accurate modeling of communications and network topology was one of the key to such achievements. Another less obvious key was the choice to combine in a single tool the possibility to do both offline and online simulation.

Index Terms—Simulation, MPI runtime and applications, Performance prediction and extrapolation.

1 Introduction

Predicting the behavior of distributed algorithms has always been a challenge, and the scale of next-generation High Performance Computing (HPC) systems will only make the situation more difficult. Consider that the Tianhe-2, the second fastest machine in the Top500 list, is made up of 16,000 compute nodes, each comprising two Intel Ivy Bridge Xeon processors and three Xeon Phi chips and that Sunway TaihuLight, the fastest machine in the Top500 list comprises 4,096 compute nodes of 260 cores each, for an astonishing total of more than 10 million cores. No human being could possibly track the activity of the individual components that make up these machines, and there is no known simple abstraction that can be used to reliably model the performance of an arbitrary algorithm running on a particular distributed system (in fact, such an abstraction is hardly possible due to its relation to the halting problem). Thus, performance modeling and software engineering for these systems increasingly require a simulation-based approach, and this need will only become more apparent with the arrival of Exascale computing by the end of the decade.

As there are currently no widely accepted standards for simulation of distributed systems, most research conclusions are based on one-off programs or scripts wherein the authors have made any number of (undocumented) simplifying assumptions. This code is not often made available to other researchers for review, which makes it difficult to validate experimental results or meaningfully compare proposed solutions. This is a complicated problem that affects disciplines outside of computing as well, and it will not be solved by a single technical solution. However, standard frameworks for modeling and simulation of HPC systems are an essential requirement for improving the situation.

Several simulators have been proposed recently in the HPC domain to fulfill this need (e.g., [1], [2], [3], [4], [5], [6]). Our previous experience [7] with grid, cloud and peer-to-peer simulators is that most tools that focus solely on scalability and performance often rely on simplistic network models that lead to wild inaccuracies when predicting the behavior of network-intensive applications. At the other extreme, microscopic models from the architecture or from the networking community are sometimes used to get perfectly accurate performance predictions about individual computer systems, but these studies are inherently limited in scale to only a small number of nodes.

When developing a simulation for any complex system it is important to consider which aspects are the most important to model, and for distributed computing platforms the most important consideration is clearly inter-node communication. Message passing is the most commonly used abstraction to program and fully exploit the capacities of current petascale systems–Implementations of the MPI standard are used on all the leading supercomputers of the Top500 list. While MPI and applications using this standard have proved to be usable on machines that comprise hundreds of thousands individual cores, scaling the specification, the implementations, and the applications to exascale remains a true challenge.

In this article, we explore the tradeoffs between scalability and accuracy that appears when studying HPC systems at extreme scale. This study is motivated by a number of typical questions facing HPC researchers and system administrators. Each question defines a broad topic area and serves as a starting point in the definition of concrete use cases for simulations of distributed computational platforms. They are as follows:

1) What would be the performance of a given application run on a hypothetical larger version of a currently available hardware platform? (i.e., **Quantitative performance extrapolation**)
2) What would be performance of a given application

- *A. Degomme is with Basel University, Switzerland.*
- *A. Legrand is with Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, F-38000 Grenoble, France.*
- *G. Markomanolis is with Supercomputing Laboratory of King Abdullah University of Science and Technology, Jeddah, Saudi Arabia.*
- *M. Quinson is with IRISA laboratory (ENS Rennes / Université de Rennes 1 / Inria / CNRS), Rennes, France.*
- *M. Stillwell is with Imperial College London, London, UK.*
- *F. Suter is with IN2P3 Computing Center, CNRS / IN2P3, Lyon - Villeurbanne, France and Inria, LIP, ENS Lyon, Lyon, France.*

when run on a hypothetical hardware platform with characteristics, e.g., type of processors, network interconnect and/or topology, that differ from those currently available? (i.e., **Qualitative performance extrapolation**)

3) Given a workload, can we detect unexpected behavior of a hardware platform from discrepancies between simulated and actual executions? (i.e., **Detection of hardware misconfiguration**)
4) Can the internals of an MPI runtime, i.e., collective communication algorithms, be finely tuned using data generated by an extensive campaign of simulations? (i.e., **MPI runtime tuning**)
5) Can simulation provide a reliable and useful environment for teaching students to program distributed algorithms, given the price and scarce availability of the real resources? (i.e., **Teaching**)

In this work, we identify a number of important features and components that should be provided by any sound simulation framework for enabling such studies of MPI applications at large scale. For each feature or component, we first present the related work and then detail the contributions we made in SMPI, a stable, open-source[1], and flexible simulator leveraging the SimGrid framework. This allows to better understand the positioning of recent MPI simulators in the cost/accuracy/applicability spectrum.

In Section 2 we present how the applications' behavior can be captured for study. Frameworks for the simulation of MPI applications generally follow one of the two complementary approaches: *off-line simulation*, or "post-mortem simulation" and *on-line simulation*, or "simulation via direct execution". In off-line simulation, a log, or trace, of MPI communication events (begin and end time-stamps, source, destination, data size) is first obtained by running the application on a *host platform*. A simulator then replays the execution of the application as if it were running on a *target platform* that may have different characteristics than the original host. In on-line simulation the application code is executed on the *host platform* that attempts to mimic the behavior of the *target platform*. During this execution part of the instruction stream, typically calls related to inter-process communication, is intercepted and passed to a simulator.

In Section 3 we discuss the problem of modeling the hardware resources of the target infrastructure in a way that is both sound and efficient. While dealing with the required scale of these simulations alone can be a challenge, oversimplification of the underlying network models that ignores many important phenomena can lead to research results of limited interest for reasoning about real-world systems. The question of the validity of the network model is often left for future work. However, we claim that given the performance evaluation challenges that need to be addressed, prediction accuracy has to be the primary goal in the design of any simulator for HPC, and that the underlying network models have to be validated (at least at small scale). The computational aspects of the simulation are equally difficult to model closely without expending considerable time and effort, but fortunately we show in this article that rough estimates based on monotonic functions of benchmark timings are often sufficient to provide reasonably accurate results.

An essential requirement for accurate simulation, detailed in Section 4, is to faithfully take the specifics of the software runtime environment into account. In the case of MPI one of the most important relevant considerations is how the collective communication operations are implemented. Many MPI applications spend a significant amount of time in such operations, that thus are a key to performance. Several algorithms exist for each collective operation, each of them exhibiting very different performance depending on various parameters such as the network topology, the message size, and the number of communicating processes [8]. MPI implementations make a careful selection at runtime that has to be captured by a simulation framework to fully model the application dynamics.

A final consideration that will be discussed in this paper is the importance of having an efficient and scalable simulation engine. As detailed in Section 5, taking contention into consideration mandates specific adaptations to the techniques that are classically used for the simulation of Discrete-Event Systems.

The remaining sections can be described briefly: Section 6 wraps up our contribution and situates it compared to the state of the art. Section 7 provides an evaluation of our contribution by exploring how SMPI could provide answers to the questions raised by the aforementioned use cases. Finally we summarize our contributions in Section 8.

2 Application Behavior Capture

As mentioned in the introduction, there exist two main approaches to capture and simulate the behavior of MPI applications. In *off-line* simulation, a trace of MPI communication events is first obtained and then replayed. In the *on-line* simulation approach the actual application code is executed and part of the instruction stream, is intercepted and passed to a simulator.

Most simulators that have been proposed follow the *off-line* simulation approach [3], [9], [10], [11], [12] while relatively few projects go for *on-line* simulation [13]. This may be due to the fact that *on-line* simulation is technically very challenging, but only mandatory for the study of dynamic network-aware applications, while most MPI applications were based on rigid communication patterns until recently. However, the irregularities inherent in modern very large scale platforms impose a requirement for dynamic applications, and this in turn can mandate the use of on-line simulation.

In this section, we first present the challenges and related work for each of these approaches. Then we detail the design and implementation choices of the SMPI framework that allows for both the off-line and on-line simulation of MPI applications.

2.1 Background on Off-line Simulation

The typical approach to *off-line simulation* is to compute the durations of the time intervals between MPI communication operations, or "CPU bursts". Upon replaying the application, the CPU bursts are modified to account for

1. Available at http://simgrid.org

the performance differential between the platform used to obtain the trace and the target platform, using either simple scaling [9], [10] or a more sophisticated convolution between the application computational signature and the target platform's hardware signature [14]. Network communications may be simulated based on the communication events recorded in the trace and a model of the network. As the size of computing platform increases (potentially towards exascale) , it is crucial that post-mortem analysis be scalable.

One problem with the off-line approach is that most tools require traces with precise and uniform measured elapsed times for all events (computation, communication) included, which limits the scalability to the largest dedicated homogeneous platform available. Additionally, the sheer number of recorded events leads to traces so large that transferring, storing, and processing them becomes an analysis bottleneck. One possible solution to this challenge is to forego recording full traces, and instead to record only aggregated statistical information. Using such lossy traces has proved sufficient for the automated or semi-automated detection of performance bottlenecks, load imbalance, and undesired behaviors. Unfortunately, this approach prevents more in-depth analysis of the application.

Another problem with the off-line approach is that in many cases one wishes to consider other platforms than the one on which the event trace has been obtained, perhaps even hypothetical platforms that are not currently available.

While trace extrapolation to larger numbers of nodes or alternative hardware configurations is feasible for some applications [3], [10], [15], this approach requires making assumptions about program behavior that may be impossible to justify in the general case.

2.2 SMPI: Time-Independent Trace Replay

To address the different scalability challenges raised by the off-line simulation of MPI applications, we proposed in [16] a trace replay framework whose main originality is to eliminate all time-related information from application event traces, using what we call "time-independent traces". For each event occurring during the execution of the application, e.g., a CPU burst or a communication operation, we log its volume (in number of executed instructions or transferred bytes) instead of the time when it begins and ends or its duration. The main advantage is that the logged information does not depend on the hardware characteristics of the platform on which the trace is collected. The size of the messages sent by an application is not likely to change according to the specifics of the network interconnect. The number of instructions performed within a basic block does not increase with the processing speed of the CPU. These two claims do not hold for adaptive MPI applications that modify their execution path according to the execution platform. But since *off-line* simulation cannot be applied to such applications anyway, time-independent traces does not reduce the application scope. The trace acquisition platform can be *composite*, e.g., a set of heterogeneous clusters interconnected via a wide-area network and the application execution can be *folded*, i.e., by running multiple MPI processes on the same compute node. Results in [16] show how using folding on a composite platform it is possible to acquire a trace for a 16,384-process execution of the LU NAS Parallel Benchmark using either 778 compute nodes aggregated from 18 geographically distant clusters or a single cluster with 20 nodes with each 24 cores.

For the acquisition of time-independent traces, we developed a lightweight instrumentation method, call MinI, that leverages the PMPI interface of the MPI standard. Initially intended for profiling and tracing purposes, this interface allows the user to attach arbitrary callback functions to MPI calls.

This code retrieves hardware performance counters, the calling parameters of MPI functions, and generates event traces in an efficient way. This approach is guaranteed to perform the minimal amount of instrumentation.

However, the main drawback of the time-independent traces is their verbosity, hence their size. As all the events are explicitly logged, the trace size grows linearly with both the problem size and the number of processes. To address this last scalability challenge, we recently modified the ScalaTrace [17] off-line analysis tool to produce compact time-independent traces that are orders of magnitude smaller than those produced by MinI [18]. Thanks to ScalaTrace these traces of near-constant size preserve the structural information and temporal event order of the original application.

2.3 Background on On-line Simulation

The first challenge in on-line simulation of MPI applications is to capture the MPI calls to be mediated in the simulator. Another big challenge can be found in the interaction between the application and the simulation kernel. One can use a distributed simulator, fold the application into a single process, or connect the distributed processes to a sequential simulator in another way. Obtaining full coverage of the MPI standard is a third challenge, as it describes hundreds of specific functions that must be implemented in simulation. Not all functions are equally important: a few dozen are used very frequently, while most others are used only seldomly. Finally, most existing projects target the study of platforms that are larger than the one at hand. This poses a fourth challenge related to resource over-subscription.

The most natural way to capture the MPI calls is to leverage the PMPI interface that is proposed by every MPI implementation. The xSim project [5] uses this capability to intercept and mediate the MPI calls through the simulator. It should be noted that concerning MPI collective operations, this approach only captures high level calls and not the specific point-to-point communications used to implement them. This reinforces the MPI standard coverage problem: every call present in the application must be correctly modeled by the tool.

The MPI calls can be captured at other points within the stack. For example, one could design a specific MPICH or OpenMPI driver that would mediate every communication through the simulator. This would, however, tie the solution to a specific MPI implementation. In BigSim [2], the application is executed on top of the Charm++ runtime (after a semi-automatic code adaptation), and the application trace is captured through the logging features of Charm++. This

elegant solution to the over-subscription challenge could be adapted to another runtime environment for online simulation. Since this captures only low-level events such as point-to-point communications while most high level calls are decomposed into a smaller set of lower level functions, this even alleviates the MPI standard coverage problem.

A third approach, followed by SST Macro [4], is to propose an ad-hoc implementation of the MPI standard. It reduces the technical difficulties posed by the PMPI approach while enabling the capture of both high-level and low-level events. A drawback is that it requires more development work to cover the MPI standard. The key challenge is then the application/simulator interactions.

To scale up the simulations, several tools opt for a folding of all simulated MPI ranks into the same address space, as threads in a single process. This design is more likely to scale than relying on full UNIX processes, but has the drawback that global variables in the MPI application will clash unless they are *privatized* to ensure that the simulated MPI ranks are still isolated from each other. In xSim, the memory area that contains the global variables is duplicated to provide a private copy to each MPI rank. In an ELF executable, global variables are stored in the `.bss` (for uninitialized variables) and `.data` (for initialized variables) memory segments. They are usually contiguous (or possibly interleaved with read-only segments) once the program is loaded into memory. When switching from the rank A to the rank B, xSim copies the process-wide global segments in the private copy of A, and then the private copy of B is copied in the process-wide areas. This approach is very robust since it leverages proven techniques, but the copy overhead can become problematic. In BigSim, the MPI application is virtualized to run on top of Charm++ to capture a trace. The globals are privatized using a trick based on the Global Offsets Table (GOT). This table is part of the dynamic linking mechanism of Linux: it contains the address of every dynamic symbol (global variable or function) that was loaded dynamically. The trick used in BigSim is to give a private copy of the GOT to each MPI rank, and to restore the actual GOT used by the system when switching between ranks. The overhead is greatly reduced since the GOT is much smaller than the actual global memory segments, but it is only applicable to dynamic symbols and not static ones. This is particularly problematic for static local variables.

2.4 SMPI: Ad-hoc Virtualization of MPI Applications

SMPI is based on a complete reimplementation of the MPI standard on top of the SimGrid simulation kernel. Even though the full MPI standard is not covered yet, SMPI can run many existing C/C++ or Fortran MPI applications unmodified. The most prominent feature sets that are still missing are Remote Memory Access, MPI I/O, the MPI-3 topology and communicator advanced functions, the rank spawning functions, and the support for multithreaded MPI applications. We ensure the conformance of the implemented functions by running the MPICH internal compliance tests as part of our automatic testing and build process. Adding the missing features is perfectly possible in the framework and only requires further development.

Since SimGrid's discrete-event simulation kernel is sequential (but fast, see Section 5), we also follow the approach consisting in folding all the MPI ranks as threads in a single process. We have explored various techniques for (semi-) automatic privatization of global variables. Early versions of SMPI relied on automatic source-to-source transformation, but this proved to be fragile. Moreover, such modification can change the performance of CPU-intensive kernels and impact the simulation accuracy. A compiler plugin would have been more robust without solving the accuracy issue. The current version of SMPI uses an approach similar to that of xSim, but is more efficient. Instead of copying the memory over, we remap it using the `mmap` system call. This leverages the virtual memory mechanism of the operating system to ensure that the address range allocated for global variables is mapped to the private memory of the currently executing rank without any performing any memory copy operations.

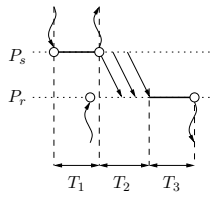
These different techniques are only possible when the MPI ranks are activated by the simulation kernel sequentially and in mutual exclusion, to avoid any memory consistency issues. Actually running the MPI ranks in their own UNIX process, and interconnecting them to a sequential simulator using some sort of Inter-Process Communication (IPC, e.g., sockets or shared memory) mechanism could alleviate this limitation, but to the best of our knowledge, no existing framework has implemented this approach.

3 Infrastructure Modeling

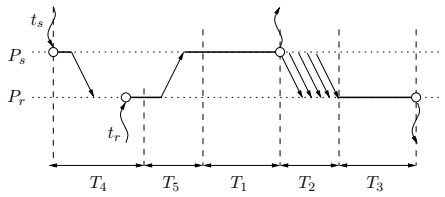
3.1 Background on Network Modeling

Packet-level simulation has long been considered the "gold standard" for modeling network communication, and is available for use in some HPC simulation frameworks [2], [20]. Such simulations reproduce the real-world communication behavior down to movements of individual packets that contain both user data and control information. However, packet-level simulations of simple network interactions often take significantly longer than the corresponding real-world events, and parallel applications are likely to generate large amounts of network traffic. Additionally, implementing packet-level simulations that accurately model real world behavior requires correctly accounting for a vast array of factors [21], including minutia such as cable length, network router model, and network driver or firmware version numbers. In practice, there is little difference between a model that is inaccurate because it has been simplified and an accurate model of the wrong system. In this context we can see that a less-accurate, but much faster, model of network communication may provide equally useful results.

When packet-level simulation becomes too costly or intractable, the most common approach is to resort to delay models that simply ignore the network complexity. Among these models, the seminal LogP model [22] captures the key characteristics of real communication platforms while remaining amenable to complexity analysis. A parallel machine is thus abstracted with four parameters: L is an upper bound on the *latency* of the network, i.e., the maximum delay incurred when communicating a word between two machines; o denotes the CPU *overhead*, i.e., the time that a processor spends processing an emission or a reception and during which it cannot perform any other operation; g is the *gap* between messages, whose reciprocal is the



(a) Asynchronous mode ($k \leq S$).



(b) Rendez-vous mode ($k > S$).

Routine	Condition	Cost
`MPI_Send`	$k \leq S$	T_1
	$k > S$	$T_4 + T_5 + T_1$
`MPI_Recv`	$k \leq S$	$\max(T_1 + T_2 - (t_r - t_s), 0) + T_3$
	$k > S$	$\max(o + L - (t_r - t_s), 0) + o + T_5 + T_1 + T_2 + T_3$
`MPI_Isend`		o
`MPI_Irecv`		o

(c) MPI routine costs.

$$T_1 = o + kO_s \qquad T_2 = \begin{cases} L + kg & \text{if } k < s \\ L + sg + (k-s)G & \text{otherwise} \end{cases} \qquad T_3 = o + kO_r \qquad T_4 = \max(L + o, t_r - t_s) + o \qquad T_5 = 2o + L$$

Figure 1: The LogGPS model [19] in a nutshell.

processor communication bandwidth for short messages; and P represents the number of processors.

The LogGPS model proposed in [19] introduces additional parameters: G to represent the larger effective bandwidth experienced by long messages and two parameters s and S to capture the lack of linearity in message size and the existence of a synchronization threshold. Overheads are represented as affine functions: $o + kO_s$ where O_s (resp. O_r) is the overhead per byte at the sender (resp. receiver) side. This model is described in Figure 1, where t_s (resp. t_r) is the time at which `MPI_Send` (resp. `MPI_Recv`) is issued. When the message size k is smaller than S, messages are sent asynchronously (Figure 1a). Otherwise, a *rendez-vous* protocol is used and the sender is blocked at least until the receiver is ready to receive the message (Figure 1b). The s threshold is used to switch from g to G, i.e., from short to long messages, in the equation. The message transmission time is thus continuously piece-wise linear in message size (Figure 1c).

While used in some HPC simulation frameworks [3], there are several characteristics of the LogGPS model that make it unsuited to modern computing infrastructures: expressing overhead and transmission times as **continuous piece-wise linear** functions of message size is not realistic at large scale; the **single-port model** unrealistically implies a processor can only be involved in at most one communication at a time; and the **lack of topology support** means that the possibility of contention on the core of the network is completely ignored. Several extensions of LogGP have thus been proposed to alleviate these flaws. For example the LoGPC model [23] accounts for average contention in the core of regular (k-D torus) networks, while the LogGPH [24] model accounts for the non uniformity of communications in hierarchical networks. More recently the τ-Lop model [25] incorporates the number of concurrent transfers directly in the transmission or synchronization costs but does not build any particular network topology.

Flow-level models provide an alternative to both expensive packet-level models and simplistic delay models. Communications, represented by *flows*, are simulated as single entities rather than as sets of individual packets. The time to transfer a message of size S between processors i and j is then given by $T_{i,j}(S) = L_{i,j} + S/B_{i,j}$, where $L_{i,j}$ (resp. $B_{i,j}$) is the end-to-end network latency (resp. bandwidth) on the route connecting i and j. Estimating the bandwidth $B_{i,j}$ is difficult as it depends on interactions between flows that occupy the same network segments and each of these flows may be influenced by others in different parts of the network. Computing message transfer times generally amounts to determining which share of the bandwidth is allocated to each flow at a given moment in time. Such models are rather flexible and account for many non-trivial phenomena (e.g., RTT-unfairness of TCP or cross-traffic interferences [7]), but have been disregarded so far for the simulation of MPI applications for at least two reasons: First, until a few years ago contention could be neglected for high-end machines. Second, such models are quite complex to implement and are often considered as too slow and complex to scale to large platforms. However, neither of these assumptions remains true today, and flow-based approaches can lead to significant improvements in simulation accuracy over classical delay models.

3.2 SMPI: A Hybrid Network Model

To capture all the relevant effects observed when executing MPI applications on a commodity cluster with a hierarchical Ethernet-based interconnection network, we proposed in [26] an original hybrid model that builds upon both LogGPS and flow-based models. All the presented experiments were conducted on the *graphene* cluster of the Grid'5000 testbed [27]. This cluster comprises 144 2.53 GHz Quad-Core Intel Xeon x3440 nodes spread across four cabinets, and interconnected by a hierarchy of 10 Gigabit Ethernet switches (see Figure 2).

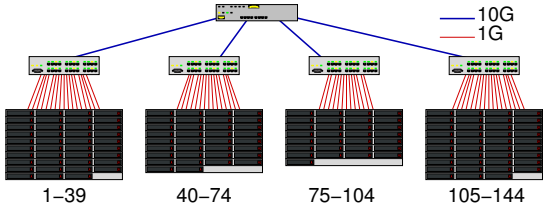


Figure 2: *graphene*: a hierarchical Ethernet-based cluster (https://www.grid5000.fr/mediawiki/index.php/Nancy:Network).

To design this hybrid model, we first ran a set of point-to-point communication experiments according to the following protocol: To avoid size and sequencing measurement bias, the message size is exponentially sampled from 1 byte to 100 MiB. We ran two "ping" and one "ping-pong" experiments. The ping experiments aimed at measuring the

time spent in the `MPI_Send` (resp. `MPI_Recv`) function by ensuring that the receiver (resp. sender) is always ready to communicate. The ping-pong experiment allowed us to measure the transmission delay. We ran our analysis on the whole set of raw measurements rather than on averaged values for each message size to prevent behavior smoothing and variability information loss. This allowed us to study the asynchronous part of MPI (from the application point of view) without any a priori assumptions on where switching may occur.

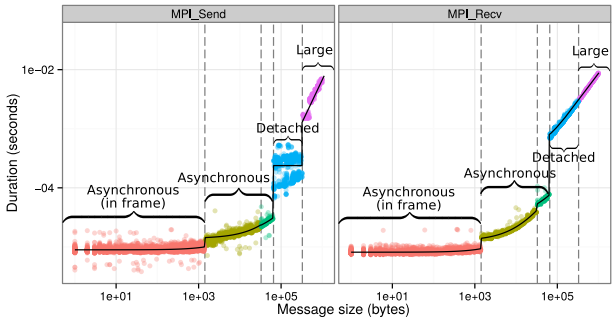


Figure 3: `MPI_Send` and `MPI_Recv` duration as a function of message size.

From the results shown in Figure 3, we identify four distinct modes:

- **Asynchronous (in frame)** (when $k \leq 1,420$): this mode corresponds to messages that fit in a Ethernet frame and are sent asynchronously by the kernel.
- **Asynchronous** (when $1,420 < k \leq 32,768$ or $32,768 < k \leq 65,536 = S_a$): these messages are still sent asynchronously but incur a slight overhead compared to smaller messages. The distinction at $k = 32,768$ does not really correspond to any particular threshold on the sender side but is visible on the receiver side where a small gap is noticed.
- **Detached** (when $65,536 < k \leq 327,680 = S_d$): this mode corresponds to messages that do not block the sender but require the receiver to post the reception before the communication actually takes place.
- **Large** (when $k > 327,680$): for such messages, both sender and receiver synchronize using a rendez-vous protocol before the data is sent. Except for the waiting time, the durations on the sender side and on the receiver side are very close.

As illustrated by Figure 3, the duration of each mode can be accurately modeled through linear regression. From these observations we derive the model described in Figure 4. It distinguishes three modes of operation: asynchronous, detached, and synchronous. Each mode can be divided in sub-modes where discontinuities are observed. The "ping" measurements are used to instantiate the values of o_s, O_S, o_r, and O_r for small to detached messages. By subtracting $2(o_r + k.O_r)$ from the round trip time measured by the ping-pong experiment, and thanks to a piece-wise linear regression, we can deduce the values of L and B. It is interesting to note that similar experiments with `MPI_Isend` and `MPI_Irecv` show that modeling their duration by a constant term o as done in [19] is not a reasonable assumption for simulation or prediction purposes.

Distinguishing these modes may be of little importance when simulating applications that only send particular message sizes. However, accurately accounting for all such peculiarities allows us to obtain good predictions in a wide range of settings, without any application specific tuning.

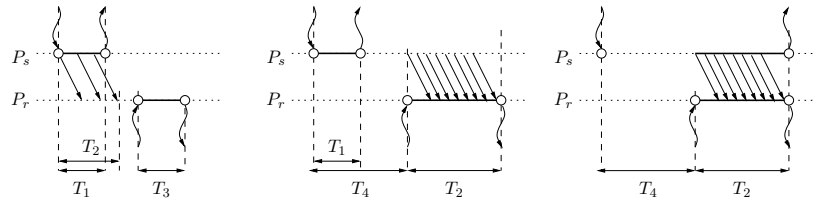
Within a cluster or cluster-like environment the mutual interactions between send and receive operations cannot safely be ignored. To quantify the impact of network contention of a point-to-point communication between two processors in a same cabinet, we artificially create contention and measure the bandwidth as perceived by the sender and the receiver. We place ourselves in the **large** message mode where the highest bandwidth usage is observed, transfer 4 MiB messages, and use concurrent `MPI_Sendrecv` transfers. We increase the number of concurrent bidirectional transfers from 1 to 19 and measure the bandwidth on the sender (B_s) and receiver (B_r) sides. A single-port model, as assumed by the LogGPS model, would lead to $B_s + B_r = B$ on average since both directions strictly alternate. A multi-port model, as assumed by flow-based models, would estimate that $B_s + B_r = 2 \times B$ since communications would not interfere with each other. However, both fail to model what actually happens, as we observe that $B_s + B_r \approx 1.5 \times B$ on this cluster.

We model this bandwidth sharing effect by enriching the simulated cluster description. Each node is provided with three links: an uplink and a downlink, so that send and receive operations share the available bandwidth separately in each direction; and a specific *limiter* link, whose bandwidth is $1.5 \times B$, shared by all the flows to and from this node. Such a value represents the effective limitation due to the card capacity and the protocol overhead and can be determined with an appropriate benchmark. This modification is not enough to model contention at the level of the entire *graphene* cluster, which is composed of four cabinets interconnected by 10 Gb links. Experiments show that these links become limiting when shared by several concurrent pair-wise communications between cabinets. This effect corresponds to the switch backplane and to the protocol overhead. We capture it by describing the interconnection of two cabinets as three distinct links (uplink, downlink, and *limiter* link). The bandwidth of this third link is set to 13 Gb as measured. The resulting topology is depicted on Figure 5.

3.3 Background on CPU Modeling

Modeling of computation so that it is possible to efficiently predict the execution time is an open problem if the target system is not available. As for network modeling, using microscopic models such as those presented in [28] is the standard approach to accurately describe CPU behavior, but with a scalability limited up to a few nodes in practice. The SST [29] can leverage cycle-accurate simulators from the architecture community, but the authors advise the use of more efficient coarse-grain CPU modeling (provided by SST Macro [4]) for studies that do not mandate exact yet expensive CPU models.

Such coarse-grain CPU models are very commonly used in the literature to simulate HPC systems. The CPU load induced by communications is usually ignored since high-quality network cards allow to offload network protocol



$(k \leq S_a)$ (a) Asynchronous mode

$(S_a < k \leq S_d)$ (b) Detached mode

$(k > S_d)$ (c) Synchronous mode

Routine	Condition	Cost
`MPI_Send`	$k \leq S$	T_1
	$k > S$	$\max(T_4, 0) + T_2$
`MPI_Recv`	$k \leq S$	$\min(t_s + T_2 - t_r, 0) + T_3$
	$k > S$	$\max(-T_4, 0) + T_2$

If $k \in I_i$:

$$\begin{cases} T_1 = o_s^{(i)} + kO_s^{(i)} & T_2 = L^{(i)} + k/B^{(i)} \\ T_3 = o_r^{(i)} + kO_r^{(i)} & T_4 = t_r - t_s \end{cases}$$

(d) Communication costs

Figure 4: The "hybrid" network model of SMPI in a nutshell.

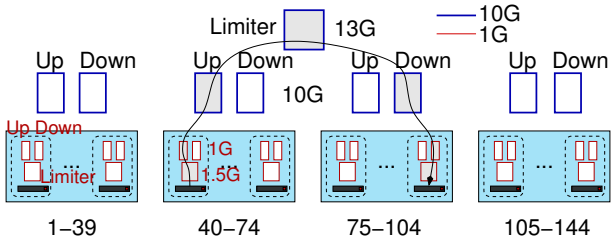


Figure 5: Modeling *graphene*: rectangles represent capacity constraints. Grayed rectangles show those involved in an inter-cabinet communication.

duties. This allows to focus on the Sequential Execution Blocks (SEBs) that each thread of the application executes between MPI primitives and it is then sufficient to predict the execution time of each SEB as a whole.

The most simplistic approach is to say that processor B is some number of times faster or slower than processor A, and then speed up or slow down the execution time of the same code on B by this factor relative to its execution time on A. In many this may give reasonable results, particularly if the architectures are similar.

However, this approach can only provide the roughest estimates across different architectures (e.g., moving from `x86_64` to POWER architecture) or across the multiple variations of a processor line (e.g., Sandy Bridge vs. Ivy Bridge Intel Xeon E5) as these differing systems will vary not only in raw speed, but across a number of different performance related elements (number of registers, number of hyperthreading cores, speed of floating point computation and number floating point computational elements, bandwidth to memory, etc.). Indeed, while the execution of a given piece of code may be fully deterministic, its exact execution time may be impossible to predict precisely without the perfect knowledge of system state that is only available to cycle-accurate simulators. Factors such as cache and pipeline effects and memory access times can make accurately predicting the runtime of a particular piece of code complicated even across multiple runs on the same system.

Several simulation frameworks try to extend the simplistic analytic models to bridge the gap with the overly expensive cycle-accurate models in a less expensive way. In BigSim [2], the user can provide a specific execution time projections for SEB, or the execution time can tentatively be devised from analytic memory models developed in [14]. Similarly, Dimemas [1] specifies how each SEB should be scaled when running on different architectures. By contrast, PSINS [9] attempts to extrapolate the performance of SEBs from hardware counters.

Such approaches become barely tractable with the complexity evolution of existing infrastructures. They will likely fail to predict the computational performance of upcoming systems present with multiple cores, implicit memory accesses, multi-level caches inducing multi-level contentions (which are sometimes handled seamlessly by the hardware using unknown algorithms) dedicated hardware accelerators, and automatic performance variations due to energy considerations. In particular, the pressure put on cache by multiple cores and the effect this has on performance seems difficult to predict accurately even with complex analytic models.

Given the relatively poor predictive power of analytic models and the poor performance of microscopic models, the Phantom project [11] does not model the CPU performance, but instead simply benchmark the CPU bursts on a processor of the target platform. The measured timings are added to the captured trace, which is then replayed offline.

3.4 SMPI: Sequential Execution Block Emulation

SMPI emulates SEBs by directly executing them on one processor of the target infrastructure whenever possible. A constant factor can be used to account for speed differences between similar infrastructures. This extends [11] by making this approach applicable to both off-line and on-line studies. It makes it possible to study applications that are network-aware (thanks to the on-line analysis) and data-dependent (thanks to the emulation).

As explained in Section 2.4, most MPI codes can be compiled and executed on top of SimGrid without requiring any modification. Such an approach is usually quite expensive in terms of both simulation time and memory requirements since the whole parallel application is actually running on a single host machine. However, it is common in the HPC context that not all the code is data-dependent. For instance, applications often have several options regarding how to perform a computation depending on the input data size, the number of available processors, or some configuration option. In such a particular case it is necessary to run through all this decision logic to faithfully capture the application behavior, but in general the execution structure is independent on the computation results. Then most computation-intensive kernels could actually be

skipped for the purpose of the simulation. To this end, SMPI provides several macros (e.g., `SMPI_SAMPLE_LOCAL` and `SMPI_SAMPLE_GLOBAL`) to annotate regular computation kernels. In simulation, such regions of code are executed a few times to obtain estimations of their cost and are skipped when enough samples have been obtained. If the duration of some code regions depends on known parameters, preliminary modeling can be done to evaluate the corresponding amount of flops that should be passed to the simulator through the `SMPI_SAMPLE_FLOPS` macro (see [30] for an illustration of this approach). Finally, since the computation results are irrelevant, SMPI provides the `SMPI_SHARED_MALLOC` and `SMPI_SHARED_FREE` macros to indicate that in simulation some data structures can safely be shared between processes (e.g., an input matrix). Making such allocations at least once is important to allow sampled computation kernels to execute correctly. Such annotations (kernel sampling and memory folding) make it possible to drastically reduce both memory footprint and simulation duration.

4 Collective Operations Modeling

4.1 Background on Collective Operations Modeling

As mentioned in the introduction, the way MPI collective operations are implemented is of utmost importance to application performance. Several solutions have been proposed in the literature.

The Dimemas framework [1] relies on the fact that all the MPI collective operations share a very regular communication pattern. The completion time of each collective operation is thus modeled by summing up to three simple analytic terms: the data *fan-in* time; the *middle operation* time; and the data *fan-out* time. For instance, the completion time of a `MPI_Gather` operation using a binomial tree is only composed of a *fan-in* time equal to $\left\lceil \frac{\log N}{\log \text{fan}_{\text{in}}} \right\rceil \times \left(\text{latency} + \frac{\text{size}}{\text{bw}}\right)$ (amount of steps $\times$ duration of each step) while a `MPI_All2all` operation has neither *fan-in* nor *fan-out* times but only a *middle operation* time of $N(N-1) \times \left(\text{latency} + \frac{\text{size}}{\text{bw}}\right)$. Such simple models cannot capture the sophistication of real collective implementations as they neglect important effects such as network contention, but seem sufficient to capture performance trends.

In [31] the authors follow a radically different approach. Each collective communication operation is automatically benchmarked on the target platform for a broad range of parameter sets and communicator geometries, without any manual analysis. The obtained timings are then simply reinjected to predict the application performance. However, this approach does not apply to performance extrapolation or the exploration of *what-if* scenarios. Furthermore a long benchmarking time is required to instantiate such a model.

The Group Operation Assembly Language (GOAL) is a domain-specific language that aims at specifying the precise communication pattern of MPI collective communication operations [32]. The LogGOPSim simulator leverages this decomposition to replace any operation with the set of corresponding point-to-point operations [3]. While interesting this approach does not capture the logic in charge of selecting the right algorithm for an operation given a parameter set. This selection is done by a code up to several thousand lines long (for each operation) in major MPI implementations.

BigSim [2] captures the decomposition of a given collective operation into point-to-point communications during the acquisition of an execution trace. The resulting trace can then be replayed in other settings since, to the best of our knowledge, the algorithm selection only depends on the message size and the communicator geometry. The traces are then more portable than those in [31] and captured in a more automatic way than in [3]. However, this approach is limited to the off-line replay of traces and cannot be leveraged for on-line simulation.

4.2 SMPI: MPI Runtime(s) Code Integration

The optimization of the performance of MPI collective operations has received a lot of attention. MPI implementations thus have several alternatives for each collective operation and select one at runtime depending on message size and communicator geometry. For instance, in OpenMPI, the `MPI_Allreduce` operation represents about 2,300 lines of code. Ensuring that any simulated run of an application uses the same or a similar implementation is thus key to simulation accuracy.

To adhere as closely as possible to the selection logic of the targeted MPI implementation, SMPI implements **all** the collective algorithms and selection logics of both OpenMPI and MPICH [33] as well as a few other collective algorithms and selectors from Star-MPI [8], the Intel vendor implementation of MPI, and MVAPICH2 [34]. SMPI currently provides more than 120 different collective algorithms, including seven SMP-aware algorithms for `MPI_Allgather` or `MPI_Allreduce`. Users can choose the selector or algorithms from the command line, allowing them to test in simulation whether replacing one algorithm by another may be beneficial or not on a particular platform/application combination.

5 Efficient Simulation Engine

5.1 Background on Discrete Event Simulation

Simulating MPI applications requires an efficient Discrete Event Simulation (DES) kernel, especially when using packet-level simulation. One tempting approach to handle such a workload could be to build upon a stock *parallel* DES (PDES) kernel. The design of such kernels has received a lot of attention in the last few decades [35] and has even been recently used in the HPC context to simulate large networks [6], [36]. The approach followed by xSim naturally leads to the use of a PDES. Every application process is associated to a Logical Process (LP) of the PDES and communicates through a virtual MPI layer implemented over the PDES. A new challenge arising from this design is that the PDES itself is often implemented using MPI, resulting in intricate implementations where developers may have difficulty keeping track of the differences between the virtual MPI used by the application and the real MPI used to implement the PDES. Added to the extra complexity posed by the oversubscription of LPs onto the physical compute cores, this results in an quite complex design. In some

sense, this complexity comes from the seemingly natural choices made for how to capture MPI calls and manage simulation/application interactions.

The performance of a DES kernel usually depends on the efficiency of its Future Event Set (FES). This data structure stores the future simulation events and highly optimizes the retrieval of the next occurring event. Many solutions were proposed in the literature beyond the classical heap [37], which all rely on the fact that the date at which future events will occur is known beforehand.

5.2 SMPI: Optimization of a Sequential DES

From our experience, simulating parallel applications offers quite limited parallelization opportunities to the simulation kernel even for relatively loosely coupled distributed systems comprising millions of entities and the benefit for tightly coupled applications is thus low [38]. We therefore decided to focus on the optimization of a *sequential* DES.

Most of simulation costs come from the application itself, which can be resource hungry, and from the evaluation of the flow-level network model. To reduce the overhead of emulating applications, we provide memory and computation annotations (Section 3.4).

Flow-level simulations are typically several orders of magnitude faster than packet-level simulations when large messages are involved, which is why we have built on top of an existing flow-level simulator that is highly optimized for network-intensive workloads on possibly complex heterogeneous topologies [39]. With such approach, computing the date at which future events will occur in a FES is not possible when the network model integrates contention. For instance, an action often ends earlier or later than expected if some competing actions are started or canceled meanwhile. The resulting workload for the FES is quite particular and requires to leverage other techniques to efficiently compute the next occurring event, as detailed in [40]. Once such optimizations have been done, the resulting workload is intrinsically sequential and hardly benefits from classical PDES approaches.

6 The SMPI Framework

The previous sections detailed the techniques used in different simulation frameworks [1], [2], [3], [4], [5], [6], including SMPI. Table 1 summarizes these techniques in terms of *application behavior capture*, *infrastructure modeling*, *collective communication modeling*, and *simulation engine*. The choices made in the design of SMPI allow us to answer to offer a unique combination of features gathered within a single unified framework. The primary target of SMPI is the fine understanding of real applications at small or medium scale. The flit-level simulation of routing protocols at extreme scale is not the kind of study that can be done with SMPI, other tools [6] being more adapted.

Figure 6 shows how the different components detailed in the previous sections are organized within this single unified framework. From top to bottom, we have the trace replay program for the off-line simulation approach (Sec. 2.2) and the API that implements the MPI-2 standard (and a subset of the MPI-3 standard) for the on-line approach (Sec. 2.4). Both are built on top of an emulated MPI runtime that also implements all the collective communication algorithms from several real runtimes, and their selection logic (Sec. 3.4 and 4.2). This layer interacts through the SimIX module with the DES kernel, called SURF (Sec. 5.2), in which the network models are implemented (Sec. 3.2).

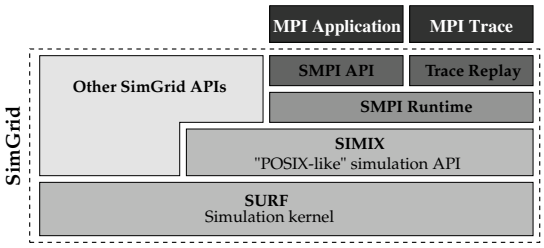


Figure 6: Internal organization of the SMPI framework.

7 Evaluation

In this section we illustrate how the on-line and off-line simulation capabilities of the proposed SMPI framework can be used to answer to several important questions related to the performance optimization of MPI applications and runtimes. All the experiments have been done either on the *graphene* cluster that was described in Sec. 3.2 (Sec. 7.3, 7.4, and 7.5) or the *Tibidado* prototype cluster (Sec. 7.2). This experimental HPC cluster, designed within the Mont-Blanc European project [41], is built using NVIDIA Tegra2 chips, each a dual-core ARM Cortex-A9 processor. The PCI Express support of Tegra2 is used to connect a 1 Gb Ethernet NIC, and the board are interconnected hierarchically using 48-port 1 Gb Ethernet switches. The descriptions of these clusters used as input to SMPI are instantiated with values obtained independently from the studied applications according to the procedure described in Sec. 3.2.

7.1 A Glimpse at Simulation Scalability

According to some recent work [5], xSim is one of the most scalable simulator of MPI applications. To illustrate this scalability, the author reports results using xSim to emulate a simple MPI program that only calls `MPI_Init()` and `MPI_Finalize()` without performing any communication or I/O operations. Using 40 nodes comprising 2×12 AMD Opteron cores, xSim allows to simulate 134,217,728 ($= 2^{27}$) MPI processes in about 1,000 seconds with 2TiB of RAM. A similar experiment with SMPI using a commodity laptop (4 core 2.10GHz Intel i7-3687U CPU) we have been able to simulate $2^{20} \approx 10^6$ MPI process with 9.1GiB of RAM in about 40 seconds (including parsing time and process creation). Given the perfect scaling we observed, simulating 2^{27} MPI processes would therefore require around 1.1TiB of RAM and last for 4,900 seconds, but using only a single core. Although machines with such amount of memory are not that commonly found for now, this limitation could be overcomed by aggregating the RAM of several nodes of a cluster through its high speed network [42].

We now present scalability results on a slightly more complicated workload. The NAS EP benchmark performs independent computations with three `MPI_Allreduce()`

Framework	Application Capture	Infrastructure Modeling			CPU	Collective	Simulation	Software
	Off-/On-line	Network Communication	Network Topology	Contention	Model	Communication Operations	Engine	Licence
BigSim [2]	Yes/No	Delay	Torus	No	Projections	Identical trace replay	DES/PDES	Charm++/Converse
CODES [6]	Yes/No	Flit level	Torus, dragonfly	No	Scaling	Tree algorithms	PDES	N/A
LogGOPSim [3]	Yes/No	LogGPS	None	No	Simple	Basic algorithms	DES	open source
Dimemas [1]	Yes/No	Delay	2 levels of hierarchy	Partial	Scaling	Analytic formula	DES	LGPL
xSim [5]	Yes/Yes	Delay	Hierarchies, torus	Yes	Emulation	Linear/tree algorithms	PDES	N/A
SST Macro [4]	Yes/Yes	Packet- or Flow-level	Torus, fat trees, dragonfly, ...	Yes	Cycle	Linear algorithms	DES/PDES	BSD
SMPI	**Yes/Yes**	**Flow-based LogGPS**	**Torus, fat trees, dragonfly, hierarchies, general graphs**	**Yes**	**Emulation**	**Algorithms and selectors from OpenMPI, MPICH,...**	**Optimized DES**	LGPL

Table 1: Summary of the Modeling approaches of various active MPI simulation frameworks.

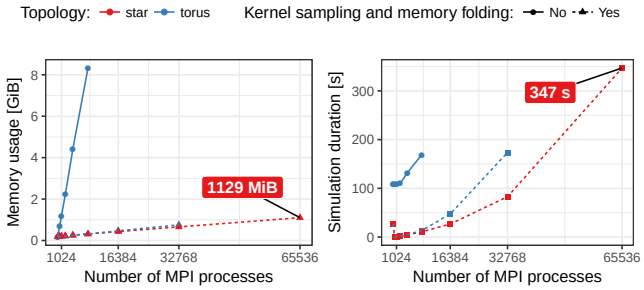


Figure 7: Simulation scalability depending on its configuration and on the workload.

operations at the end to check the correctness of the results. The performance of an emulation of unmodified EP (class B) on a $32 \times 32 \times 32$ torus is reported in Figure 7 (blue solid line) and is obviously limited by the memory consumption since all the operations and memory allocations are done. With such an approach, filling the platform with 65,536 process would require 65GiB of memory and take several hours. However, performing all computations is typically useless for performance evaluation purposes. A 6-line annotation of the code allows us to activate kernel sampling and memory folding and to drastically reduce computation time and memory consumption (blue dashed line). The time spent running the MPI code then drops from about 100 seconds to less than 1 second in all settings and the memory required for 65,536 processes drops to about 1.2GiB of RAM. The remaining of the simulation time mostly corresponds to the time spent simulating the three `MPI_Allreduce` operations. We configured our simulation to use the classical *recursive doubling* algorithm, which is not aware of the torus topology and incurs contention (the corresponding butterfly communication pattern incurs $N/2$ concurrent communications at each stage of the butterfly), hence a quadratic simulation time. A simpler star topology leads to a simpler contention pattern, hence to a shorter (although still not linear since there are $N.\log_2(N)$ communications in total) simulation time (red dashed line). Note that the lazy update mechanism used in SimGrid [39] exploits locality so that simulations of SMP- or topology- aware collective algorithms are naturally more scalable. In the rest of this section, the time needed to perform any of the simulation is always beyond a dozen minutes on a commodity machine.

7.2 Performance Prediction

The Mont-Blanc project [41] aims to assess the potential of low-power embedded components based clusters to address future Exascale HPC needs. One of the objectives of the Mont-Blanc project is thus to develop prototypes of HPC clusters using low power commercially available embedded technology such as ARM processors and Ethernet technologies. In order to start evaluating applications before 2014, a small experimental cluster of ARM systems on chip, named Tibidabo and hosted at the Barcelona Supercomputing Center, was built.

We used Tibidabo to evaluate the scalability on such architecture of BigDFT, an open-source Density Functional Theory massively parallel electronic structure code [43], which is able to scale particularly well on supercomputers. For our experiments, we disabled the OpenMP and GPU extensions at compile time to study behaviors related to MPI operations. BigDFT alternates between regular computation bursts and important collective communications. Moreover the set of collective operations that is used may completely change depending on the instance, hence the need for on-line simulation. In the following experiments, we used MPICH 3.0.4 [33] and while this application can be simulated by SMPI without any modification to the source code, its large memory footprint would require an improbably large amount of RAM if run on a single machine. By applying the memory folding and shadow execution techniques mentioned in Section 3.4 and detailed in [44], we were able to simulate the execution of BigDFT with 128 processes, whose peak memory footprint is estimated to 71 GiB, on a single laptop using less than 2.5 GiB of memory. This memory footprint compression could be further improved with additional manual annotations to allow the same laptop to support some even larger applications.

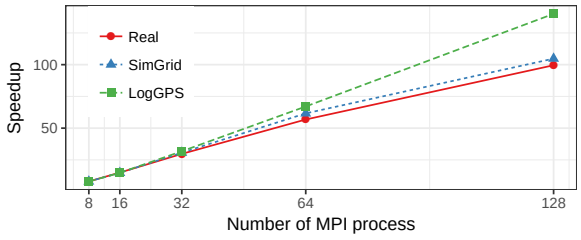


Figure 8: Prediction of the performance of BigDFT on a prototype ARM-based cluster.

Figure 8 shows the comparison of the speedup evolution as measured on *Tibidabo* for a small instance. This instance has a relatively low communication to computation ratio (around 20% of time is spent communicating when using 128 nodes and the main used operations are `MPI_Alltoall`, `MPI_Alltoallv`, `MPI_Allgather`, `MPI_Allgatherv` and `MPI_Allreduce`). This instance is thus particularly difficult to model and is expected to have a limited scalability, which one may want to observe in simulation first to avoid wasting resources or to assess the relevance of upgrading hardware. As expected, the *LogGPS*

model predicts an over-optimistic perfect scaling whereas the *flow-based* SimGrid model succeeds in accounting for the slowdown incurred by the hierarchical and irregular network topology of this prototype platform. To further demonstrate the usability of our tool, we want to mention that simulating 64 nodes of *Tibidabo*, which is made of slow ARM processors, on a Xeon 7460 with partial on-line simulation takes only half the time as running the code for real (10 vs. 20 minutes).

7.3 Performance Extrapolation

An interesting use of simulation is the exploration of "what-if" scenarios to obtain indicators of the performance of an MPI application on hypothetical hardware configurations. In this section, we illustrate this feature by analyzing the quantitative extrapolation of the target platform. This consists in simulating the execution of an application on more compute nodes than available on the actual cluster. Such a study can be conducted following either the off-line or on-line simulation approach, as both are supported by the SMPI framework.

We consider the *off-line* simulation of various instances of the LU NAS Parallel Benchmark. We compare a simulated execution based on time-independent traces [16] to a real execution on the *graphene* cluster. While we can obtain results on the actual cluster, which is made of 144 compute nodes, only up to 128 processes, we can run simulations for 16 times larger platforms (an hypothetical version of *graphene* with 2,048 compute nodes). Such extrapolation is made possible by the time-independent nature of the traces that do not require a machine at scale to be acquired.

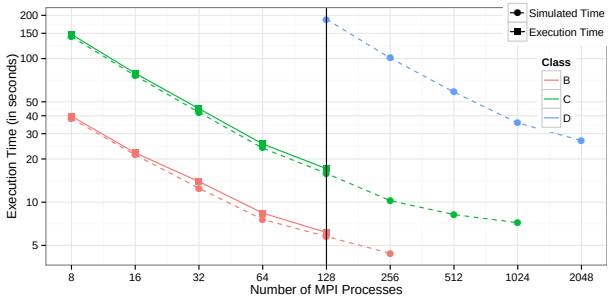


Figure 9: Simulated and actual execution times for class B, C and D LU benchmark vs. number of processes.

Figure 9 shows the obtained execution times vs. number of MPI processes. We observe that simulation follows a trend that is very coherent with the evolution of the execution time from 8 to 128 processes. For class D instances, there is no execution to compare to simulation. However, the increase in simulated time is consistent with the increased size of the data processed which is sixteen times larger than for class C instances. Moreover, the evolution of the execution time when the number of processes increases is coherent with what was measured for classes B and C. While this experiment is at a relatively modest scale, the achieved results are encouraging with regards to the capacity of the SMPI framework to simulate a larger cluster in a scalable way, regardless of the maximum size of the physical cluster at hand.

7.4 Detection of Hardware Misconfiguration

During the long series of accuracy evaluation experiments we conducted, it already happened that we encountered a significant difference between simulation and reality and these mismatches motivated some of our developments (e.g., the careful modeling of point-to-point communications for arbitrary message sizes in Section 3.2 or of collective operations evoked in Section 4.2). As a result, the SMPI network model has become very reliable and accurate and in all our studies the largest error in term of speedup or time prediction is of at most 5% and does not seem to depend on the platform size. This allows to to use SMPI beyond its primary role of a performance evaluation.

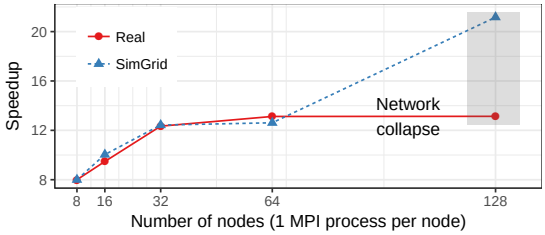
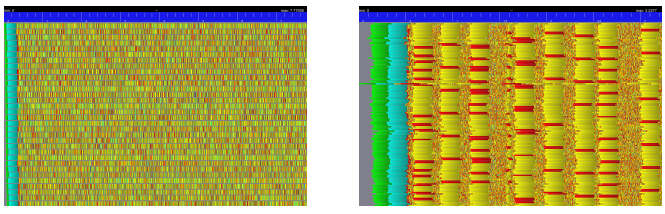


Figure 10: Evolution of the simulated and actual speedups for the CG benchmark (class B) on the *graphene* cluster.

Figure 10 shows the speedup as measured on the *graphene* cluster and as obtained with the SimGrid model with the class B of instances of the CG NAS Parallel Benchmark. This benchmark has a complex communication scheme made of a large number of point-to-point transfers of small messages. Moreover, processes are organized in a hierarchy of groups. At each level communications occur within a group and then between groups. This benchmark is then very sensitive to the mapping of the MPI processes on to the physical processors with regard to network organization, particularly in non-homogeneous topologies.

We can see that the hybrid model of SMPI leads to excellent estimations except on the scenario with 128 processes. To determine the cause of such a discrepancy between the simulated and observed times, we compare the Gantt charts of two actual runs. Figure 11 details the execution of the CG benchmark with 32 (Figure 11a) and 128 (Figure 11b) MPI processes.



(a) CG –Class B w/ 32 proc. (b) CG – Class B w/ 128 proc.

Figure 11: Illustration of a network collapse due to a TCP misconfiguration.

The execution with 128 processes shows several outstanding zones of `MPI_Send` (in red) and `MPI_Wait` (in yellow). Such operations typically take few microseconds to less than a millisecond, yet here they take 200 milliseconds.

We theorize that, due to high congestion, the switch drops packets and slows (at least) one process to the point where it stops sending until a timeout of 200 milliseconds is reached. Because of the communication pattern of the CG benchmark, blocking one process impacts all the other processes, hence the phenomenon in Figure 11b. This issue is likely to be related to the *TCP incast problem* [45], and the TCP retransmission timeout, which is equal to 200 milliseconds by default in Linux.

It is interesting to note that if we subtract these undesired delays from the measured execution time we obtain a value extremely close (within a few percents) to the simulated time given by SMPI. If we managed to fix this undesired TCP behavior inducing network collapse, our prediction would thus be quite faithful. Note that the speedup prediction has a quite surprising shape since it shows a linear increase from 8 to 32 followed by a plateau between 32 and 64 and finally a linear increase from 64 to 128. This non-convex shape can be explained by the hierarchical structure of the graphene cluster and by the fact that network load is better balanced when using the whole cluster than when restricting to only two cabinets.

7.5 MPI Runtime Tuning

As explained in Section 4, the optimization of the performance of MPI collective operations has received a lot of attention [8], [32], [46] and to ensure a faithful modeling, we have implemented in SMPI all the collective algorithms and selection logics of several standard MPI implementations. This effort allowed for better understanding of the differences between these implementations, and let us to consider whether the default choices for a given application on a given machine are optimal. As an illustration we investigate the performance of the `MPI_Alltoall` operation on a hypothetical version of the *graphene* cluster with 4 cabinets of 256 nodes each. Since we had no a priori on which region of the parameter space was the most interesting, we sampled the message size in [32 kB; 2 MB] and the number of nodes in [1; 1,024] and test each of the 15 implementations of this collective operation. Some of them being simple variations on a theme and thus having very similar performance, we only report the performance of the 10 most interesting traces in Figure 12. Durations are normalized by the message size.

First, it is easy to see that for most algorithms, the performance of the collective operation is linear with the number of processes and that a sharp behavior change can be observed when using more than a single cabinet (above 256 nodes). The graphs on the left hand-side thus show a zoom on the single cabinet scenario. Three groups of collective operations can then be distinguished. Some (2dmesh, 3dmesh, and rdb) are clearly inappropriate for such topology and have very bad performance while two others (basic_linear and bruck) achieve systematically optimal and stable performance. Interestingly, the method selected by all MPI implementations only achieves near optimal performance. In most scenarios the performance is excellent but for some combinations of message size and number of processes, it can be up to twice as bad as what could be achieved with the bruck algorithm. Although such an outcome may have easily been forecast by an expert

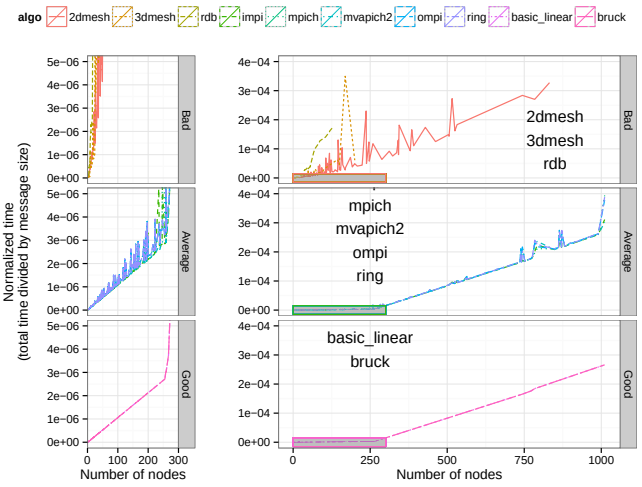


Figure 12: Evaluation of various algorithms for the `MPI_Alltoall` operation.

on such algorithms, we think it is not that trivial even for an advanced MPI user. We thus contend that SMPI-based simulation could allow typical users to determine reliably and at low cost which MPI configuration is the best for a given application on a given machine.

7.6 Using SMPI in Teaching Contexts

Teaching MPI programming to post-graduate students is both very important to avoid any future shortage of specialists, and very challenging if the students need to be granted an access to HPC systems. Indeed relying on real parallel platforms is often inadequate for "beginners". First, this teaching usage competes with the usual production workloads and misguided experimentation from students often disrupts these systems. Second, as an instructor, it is often very tempting to build on the student's observations and wish they could experiment on a machine with a better network, more nodes, a different topology, or more heterogeneous nodes. This would undoubtedly allow the students to better understand some principles, but reminding them they have only access to limited resources is not great pedagogy. Another issue issue is that transient network problems and hardware failures can negatively affect the performance of applications, clouding the intended lesson. As a final consideration, often before an assignment is due students will compete for the machine with other users and with each other, which leads to frustration if the machine is small or quite busy.

Relying on a simulator presents many advantages to teach HPC systems to post-graduate students [47]. Students do not have to experience all the burden of using a real platform, which facilitates their education. Simulations are repeatable, reliable, and fast and allow students to explore many scenarios, which is a huge pedagogical advantage. SMPI is now used in a curriculum at University of Hawai'i at Manoa. One of the typical assignments[2] consists in implementing several broadcast algorithms, using SMPI to evaluate their performance on several topologies, and compare

2. https://simgrid.github.io/SMPI_CourseWare/

them with classical implementations. In such context, the on-line simulation capability of SMPI is extremely precious as students can run all these experiments directly on their own laptop without having to resort at any time to a parallel computer (unlike what would be required with off-line simulation). Being able to easily change network parameters and topology helps students grasp the different tradeoffs at stake. The native tracing capability of SMPI and its ability to also trace the internals of collective operations was also reported to be very helpful.

8 Conclusion

In this paper we have argued that the complexity of current and next-generation large-scale distributed systems mandates a simulation-based approach to modeling the behavior of these systems and the algorithms that run on them. We proposed a number of use-cases that should be supported by a unified system for simulating distributed systems, and discussed the problems inherent in creating computationally efficient simulations that accurately model real-world behavior. Finally, we showed how SMPI supports these use cases, validated the network models used by SMPI through a series of experiments, and discussed the similarities and differences between SMPI and a number of competing projects for simulating distributed systems.

We left other HPC use cases that are not strictly related to the SMPI framework out of the scope of this paper. We refer the reader interested in the use of SimGrid for the formal verification of HPC applications to [48] or in the support of higher-level HPC runtimes for hybrid (multi-core and multi-GPUs) platforms to [49]. Simulating applications leveraging both MPI and CUDA at a larger scale than what is presented in this article is part or our ongoing work.

Acknowledgments

The authors would like to thank the SimGrid team members and collaborators who contributed to SMPI: P. Bedaride, H. Casanova, P.-N. Clauss, F. Desprez, S. Genaud, A. Gupta, and B. Videau. This work is partially supported by the Hac Specis Inria Project Lab, the ANR SONGS (11-ANR-INFR-13), CNRS PICS N° 5473, and European Mont-Blanc (EC grant 288777) projects. Experiments were carried out on a PRACE (EC grants RI-261557 and RI-283493) prototype and the Grid'5000 experimental testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and other Universities and organizations.

References

[1] R. M. Badia, J. Labarta, J. Giménez, and F. Escalé, "Dimemas: Predicting MPI Applications Behaviour in Grid Environments," in *Proc. of the Workshop on Grid Applications and Programming Tools*, Jun. 2003.

[2] G. Zheng, G. Kakulapati, and L. Kale, "BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines," in *Proc. of the 18th IPDPS*, 2004.

[3] T. Hoefler, C. Siebert, and A. Lumsdaine, "LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model," in *Proc. of the LSAP Workshop*, Jun. 2010, pp. 597–604.

[4] C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo, "A simulator for large-scale parallel architectures," *International Journal of Parallel and Distributed Systems*, vol. 1, no. 2, pp. 57–73, 2010, http://dx.doi.org/10.4018/jdst.2010040104.

[5] C. Engelmann, "Scaling To A Million Cores And Beyond: Using Light-Weight Simulation to Understand The Challenges Ahead On The Road To Exascale," *FGCS*, vol. 30, pp. 59–65, Jan. 2014.

[6] M. Mubarak, C. D. Carothers, R. B. Ross, and P. H. Carns, "Enabling parallel simulation of large-scale hpc network systems," *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[7] P. Velho, L. Schnorr, H. Casanova, and A. Legrand, "On the Validity of Flow-level TCP Network Models for Grid and Cloud Simulations," *ACM Transactions on Modeling and Computer Simulation*, vol. 23, no. 4, p. 23, Oct. 2013.

[8] A. Faraj, X. Yuan, and D. Lowenthal, "STAR-MPI: self tuned adaptive routines for MPI collective operations," in *Proc. of the 20th ACM Intl. Conf. on Supercomputing*, 2006, pp. 199–208.

[9] M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely, "PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications," in *Proc. of the 15th International EuroPar Conference*, ser. LNCS, vol. 5704, Delft, Netherlands, Aug. 2009, pp. 135–148.

[10] A. Núñez, J. Fernández, J.-D. Garcia, F. Garcia, and J. Carretero, "New Techniques for Simulating High Performance MPI Applications on Large Storage Networks," *Journal of Supercomputing*, vol. 51, no. 1, pp. 40–57, 2010.

[11] J. Zhai, W. Chen, and W. Zheng, "PHANTOM: Predicting Performance of Parallel Applications on Large-Scale Parallel Machines Using a Single Node," in *Proc. of the 15th ACM SIGPLAN PPoPP Symp.*, 2010, pp. 305–314.

[12] M.-A. Hermanns, M. Geimer, F. Wolf, and B. Wylie, "Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications," in *Proc. of the 17th Euromicro Intl. Conf. on Parallel, Distributed and Network-based Processing*, 2009, pp. 78–84.

[13] G. Zheng, S. Negara, C. L. Mendes, E. R. Rodrigues, and L. Kale, "Automatic Handling of Global Variables for Multi-threaded MPI Programs," in *Proc. of the 17th ICPADS*, 2011, pp. 220–227.

[14] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A Framework for Performance Modeling and Prediction," in *Proc. of the ACM/IEEE Conference on Supercomputing*, Baltimore, MA, Nov. 2002.

[15] L. Carrington, M. Laurenzano, and A. Tiwari, "Inferring large-scale computation behavior via trace extrapolation," in *Proc. of the Workshop on Large-Scale Parallel Processing*, 2013.

[16] H. Casanova, F. Desprez, G. S. Markomanolis, and F. Suter, "Simulation of MPI Applications with Time-Independent Traces," *CCPE*, vol. 27, no. 5, p. 1145–1168, Apr. 2015.

[17] X. Wu and F. Mueller, "Elastic and Scalable Tracing and Accurate Replay of Non-Deterministic Events," in *Proc. of the 27th ACM Intl. Conference on Supercomputing*, Eugene, OR, 2013, pp. 59–68.

[18] H. Casanova, A. Gupta, and F. Suter, "Toward More Scalable Off-Line Simulations of MPI Applications," *Parallel Processing Letters*, vol. 25, no. 3, p. 1541002, Sep. 2015.

[19] F. Ino, N. Fujimoto, and K. Hagihara, "LogGPS: a Parallel Computational Model for Synchronization Analysis," in *Proc. of the 8th ACM SIGPLAN PPoPP Symp.*, 2001, pp. 133–142.

[20] B. Penoff, A. Wagner, M. Tüxen, and I. Rüngeler, "MPI-NeTSim: A Network Simulation Module for MPI," in *Proc. of the 15th ICPADS*, Shenzen, China, 2009.

[21] G. F. Lucio, M. Paredes-farrera, E. Jammeh, M. Fleury, and M. J. Reed, "OPNET Modeler and ns-2: Comparing the Accuracy of Network Simulators for Packet-Level Analysis Using a Network Testbed," in *Proc. of the 3rd WEAS International Conference on Simulation, Modelling and Optimization*, 2003, pp. 700–707.

[22] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," in *Proc. of the 4th ACM SIGPLAN PPoPP Symp.*, 1993, pp. 1–12.

[23] C. A. Moritz and M. I. Frank, "LoGPC: Modeling Network Contention in Message-Passing Programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, pp. 404–415, 2001.

[24] L. Yuan, Y. Zhang, Y. Tang, L. Rao, and X. Sun, "LogGPH: A Parallel Computational Model with Hierarchical Communication Awareness," in *Proc. of the 13th IEEE International Conference on Computational Science and Engineering*, 2010, pp. 268–274.

[25] J.-A. Rico-Gallego and J.-C. Díaz-Martín, "τ-Lop: Modeling Performance of Shared Memory MPI," *Parallel Computing*, vol. 46, 2015.

[26] P. Bedaride, A. Degomme, S. Genaud, A. Legrand, G. Markomanolis, M. Quinson, M. Stillwell, Lee, F. Suter, and B. Videau, "Toward Better Simulation of MPI Applications on Ethernet/TCP Networks," in *Proc. of the 4th Intl. Workshop on Performance Modeling, Benchmarking and Simulation*, ser. LNCS, vol. 8551. Denver, CO: Springer, Nov. 2013, pp. 158–181.

[27] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lantéri, J. Leduc, N. Melab, G. M. andR. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche, "Grid'5000: a Large Scale and Highly Reconfigurable Experimental Grid Testbed," *IJHPCA*, vol. 20, no. 4, pp. 481–494, 2006.

[28] E. León, R. Riesen, and A. Maccabe, "Instruction-Level Simulation of a Cluster at Scale," in *Proc. of the International Conference for High Performance Computing and Communications*, Portland, OR, 2009.

[29] A. Rodrigues, K. Hemmert, B. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Riesen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob, "The Structural Simulation Toolkit," *SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.

[30] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, and B. Videau, "Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers," in *Proc. of the 21st ICPADS*, 2015.

[31] D. Grove and P. Coddington, "Communication Benchmarking and Performance Modelling of MPI Programs on Cluster Computers," *Journal of Supercomputing*, vol. 34, no. 2, pp. 201–217, 2005.

[32] T. Hoefler, C. Siebert, and A. Lumsdaine, "Group Operation Assembly Language - A Flexible Way to Express Collective Communication," in *Proc. of the 38th ICPP*, 2009.

[33] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH," *IJHPCA*, vol. 19, no. 1, pp. 49–66, 2005.

[34] D. Panda, K. Tomko, K. Schulz, and A. Majumdar, "The MVAPICH Project: Evolution and Sustainability of an Open Source Production Quality MPI Library for HPC," in *Intl. Workshop on Sustainable Software for Science: Practice and Experiences*, 2013.

[35] R. Fujimoto, *Parallel & Distributed Simulation Systems*. Wiley, 2000.

[36] M. Mubarak, C. D. Carothers, R. Ross, and P. Carns, "Modeling a Million-Node Dragonfly Network Using Massively Parallel Discrete-Event Simulation," *SC Companion*, pp. 366–376, 2012.

[37] W. T. Tang, R. S. M. Goh, and I. L.-J. Thng, "Ladder Queue: An O(1) Priority Queue Structure for Large-scale Discrete Event Simulation," *ACM TOMACS*, vol. 15, no. 3, pp. 175–204, Jul. 2005.

[38] M. Quinson, C. Rosa, and C. Thiéry, "Parallel simulation of peer-to-peer systems," in *Proc. of the 12th IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing*, Ottawa, Canada, 2012.

[39] B. Donassolo, H. Casanova, A. Legrand, and P. Velho, "Fast and Scalable Simulation of Volunteer Computing Systems Using SimGrid," in *Proc. of the Workshop on Large-Scale System and Application Performance*, Chicago, IL, Jun. 2010.

[40] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, 2014.

[41] "Mont-Blanc: European Approach Towards Energy Efficient High Performance," http://www.montblanc-project.eu/.

[42] S. Liang, R. Noronha, and D. K. Panda, "Swapping to remote memory over infiniband: An approach using a high performance network block device," in *Proc. of the IEEE International Conference on Cluster Computing*, 2005.

[43] L. Genovese, A. Neelov, S. Goedecker, T. Deutsch, S. A. Ghasemi, A. Willand, D. Caliste, O. Zilberberg, M. Rayson, A. Bergman, and R. Schneider, "Daubechies Wavelets as a Basis Set for Density Functional Pseudopotential Calculations," *Journal of Chemical Physics*, vol. 129, no. 014109, 2008.

[44] P.-N. Clauss, M. Stillwell, S. Genaud, F. Suter, H. Casanova, and M. Quinson, "Single Node On-Line Simulation of MPI Applications with SMPI," in *Proc. of the 25th IEEE Intl. Parallel and Distributed Processing Symposium*, Anchorage, AK, 2011.

[45] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP Incast Throughput Collapse in Datacenter Networks," in *Proc. of the 1st ACM workshop on Research on Enterprise Networking*, 2009, pp. 73–82.

[46] M. Banikazemi, J. Sampathkumar, S. Prabhu, D. Panda, and P. Sadayappan, "Communication Modeling of Heterogeneous Networks of Workstations for Performance Characterization of Collective Operations," in *Proc. of the 8th Heterogeneous Computing Workshop*, 1999, pp. 125–133.

[47] G. Zarza, D. Lugones, D. Franco, and E. Luque, "An Innovative Teaching Strategy to Understand High-Performance Systems through Performance Evaluation," in *Proc. of the International Conference on Computational Science*, vol. 9, 2012, pp. 1733–1742.

[48] M. Guthmuller, M. Quinson, and G. Corona, "System-level State Equality Detection for the Formal Dynamic Verification of Legacy Distributed Applications," in *Formal Approaches to Parallel and Distributed Systems - Special Session of Parallel, Distributed and network-based Processing*, Turku, Finland, 2015.

[49] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut, "Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures," *Concurrency and Computation: Practice and Experience*, May 2015.

Augustin Degomme is a senior engineer at University of Basel since 2015 and before this he worked within the Grenoble Informatics Laboratory, France since 2009. He has been working on SimGrid and SMPI frameworks since 2012, providing support for users and developing new features. He obtained his Master of Engineering in Computer Science from the National Institute of Applied Sciences in Rennes, France, in 2008.

Arnaud Legrand is a tenured CNRS researcher at Grenoble University, France since 2004. His research interests encompass the study of large scale distributed systems, theoretical tools (scheduling, combinatorial optimization, and game theory), and performance evaluation, in particular through simulation. He obtained his M.S. and Ph.D. from the Ecole Normale Supérieure de Lyon, France in 2000 and 2003, and his Habilitation Thesis in 2015 from Grenoble University, France.

George S. Markomanolis is a Computational Scientist at KAUST Supercomputing Laboratory His research is on performance evaluation and optimization of applications with a focus on weather models and support to the users of supercomputers. He obtained his M.S. from the department of Informatics and Telecommunication, National and Kapodistrian University of Athens, Greece in 2008 and his Ph.D. from the Ecole Normale Supérieure de Lyon, France in 2014.

Martin Quinson is a Professor at Ecole Normale Supérieure de Rennes since 2015. Before he was an assistant professor at the Université de Lorraine. His research is on experimentation methodologies, both through the simulation of distributed applications and through the formal assessment of distributed algorithms. He obtained his M.S. and Ph.D. from the Ecole Normale Supérieure de Lyon respectively in 2000 and 2003, and his Habilitation Thesis in 2013 from the Université de Lorraine, France.

Mark Stillwell is a Site Reliability Engineer at Cisco Meraki. Formerly a researcher at both Imperial College London and ENS Lyon, the main subjects of his research are scheduling algorithms for distributed systems and simulation of distributed computing platforms, with a focus on high performance and scientific computing. He obtained M.S. degrees in Mathematics and Computer Science from the University of Florida, USA in 2002 and 2003, and his Ph.D. from the University of Hawai'i at Mānoa, USA in 2010.

Frédéric Suter is a tenured CNRS researcher at the IN2P3 Computing Center in Lyon, France, since 2008. His research interests include scheduling, Grid computing and platform and application simulation. He obtained his M.S. from the Université Jules Verne, Amiens, France, in 1999, his Ph.D. from the Ecole Normale Supérieure de Lyon, France, in 2002 and his Habilitation Thesis from the Ecole Normale Supérieure de Lyon, France in 2014.